UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Oliver Vainumäe

3D Comic Rendering

Bachelor's Thesis (9 ECTS credits)

Supervisors: Raimond-Hendrik Tunnel, MSc
Santiago Montesdeoca, PhD

Tartu 2020

3D Comic Rendering

Abstract:

In this thesis a non-photorealistic rendering algorithm is devised for rendering a 3D scene. The algorithm follows the style of the illustrations in the *La mémoire de l'eau (Water Memory)* graphic novel. Implementation is carried out in Maya Non-Photorealistic Rendering Framework (MNPR), which is a framework for the 3D computer graphics software Autodesk Maya. The devised algorithm consists of edge detection using difference of Gaussians for stylized lines and color smudging. The output of the devised algorithm is compared to the source material and potential future improvements are offered.

Keywords: Computer graphics, non-photorealistic rendering, Autodesk Maya, MNPR, drawing lines, color smudging, hatching, edge-detection.

CERCS: P170 Computer science, numerical analysis, systems, control

3D koomiksite renderdamine

Lühikokkuvõte:

Käesolevas bakalaureusetöös loodi algoritm 3D stseenide mittefotorealistlikuks renderdamiseks. Algoritm järgib illustratsioonide stiili graafilisest romaanist nimega *La mémoire de l'eau (Veemälu).* Implementatsioon tehakse Maya Non-Photorealistic Rendering Framework'is (MNPR), mis on raamistik 3D arvutigraafika tarkvarale Autodesk Maya. Algoritmis kasutatakse Gaussi äärejoonte leidmise meetodit joonte leidmiseks ja värvide määrimiseks. Lõpuks võrreldakse planeeritud algoritmi väljundit lähtematerjaliga ning pakutakse välja potentsiaalsed edasiarendused.

Võtmesõnad: Arvutigraafika, mittefotorealistlik renderdamine, Autodesk Maya, MNPR, joonistatud jooned, värvide määrimine, joonitus, ääretuvastus.

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Table of Contents

1 Introduction

Computer graphics algorithms are used to generate images that range from pure photorealistic to stylized and non-photorealistic. There is no clear border between photorealistic and non-photorealistic rendering (NPR). Non-photorealistic renders are mostly considered to be *artistic media simulations*, which resemble some traditional art styles such as watercolor paintings, charcoal drawings[1], comic book illustrations [1]. Often those stylized renders emphasize visuals based on human perception or impressions (*eg* bold object outlines) [2]. Meanwhile a lot of computer graphics applications utilize photorealistic rendering principles (*eg* physically based rendering) to create stylized or non-photorealistic images.

Over the past few decades, advancements in the field of computer graphics have also affected non-photorealistic rendering. A variety of styles are now rendered in real-time. Real-time rendering has enabled the use of NPR in video games such as *Ōkami*[2] (2006) (see Figure 1), *Last Day of June*[3] (2017) and in the video games of the *Borderlands* series[4]. Available styles have been used to produce comics[5] and animated movies such as *Spider-Man: Into the Spider-Verse*[6] (2018) (see Figure 2). Non-photorealistic rendering is also used to generate architectural[7] or technical[8] illustrations.

Figure 1. *Ōkami*

Figure 2. *Spider-Man: Into the Spider-Verse*

[1] https://en.wikipedia.org/wiki/Charcoal_(art)
[2] https://en.wikipedia.org/wiki/%C5%8Ckami
[3] https://en.wikipedia.org/wiki/Last_Day_of_June
[4] https://en.wikipedia.org/wiki/Borderlands_(series)
[5] *Using Blender for Comics* https://youtu.be/pswYVl_xRoQ
[6] *Classic Cartoon Animation Style with a 3D Twist* https://youtu.be/tc8sLyyjHBg
[7] https://en.wikipedia.org/wiki/Architectural_rendering
[8] https://en.wikipedia.org/wiki/Technical_illustration

There are numerous tools that render non-photorealistic visuals from 3D geometry. Some of the tools are for example Jot[9] (See Figure 3), Freestyle[10], Pencil+[11], MNPR[12] and MNPRX[13]. Both MNPR (Maya Non-Photorealistic Rendering Framework) and MNPRX (a commercial extension of MNPR) are frameworks for Autodesk Maya [3].

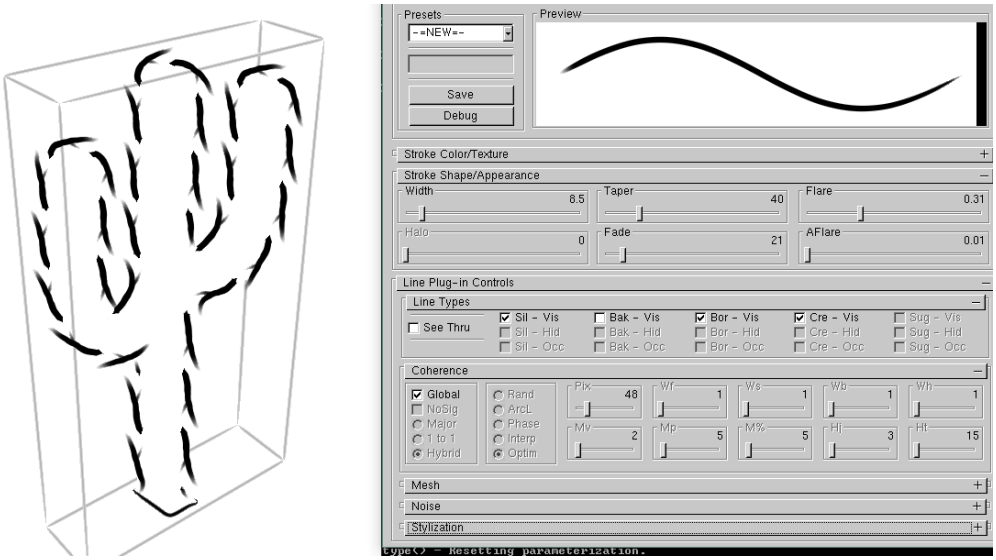


Figure 3. User interface of Jot

Autodesk Maya[14] is a popular 3D computer graphics software, which is used for modeling, simulation and rendering. The MNPR and MNPRX frameworks provide users with different real-time rendering styles and control parameters (for art-direction) over the stylized visuals. MNPR was the result of the doctoral work by Santiago Montesdeoca [3] and MNPRX is developed by Artineering – a computer graphics company in Tallinn, Estonia.

While MNPR and MNPRX are capable of rendering a variety of different styles, there is still room for more styles, which cannot be achieved with existing tools. In this thesis an algorithm is devised that allows a scene to be rendered in the style of the graphic novel *La mémoire de l'eau (Water Memory)* in real-time. Such an algorithm is of interest to Artineering. During the work we have collaborated often to implement the proposed improvements in the MNPR framework.

[9] https://jot.cs.princeton.edu/
[10] https://docs.blender.org/manual/en/latest/render/freestyle/index.html
[11] https://www.psoft.co.jp/en/product/pencil/3dsmax/
[12] https://mnpr.artineering.io/
[13] https://artineering.io/software/MNPRX/
[14] https://www.autodesk.com/

Chapter 2 describes the illustration style of *Water Memory* and points out the main stylistic elements which this algorithm needs to implement. An overview is given of how Autodesk Maya and MNPR work. Chapters 3 and 4 focus on the implementation of the algorithm based on the main aspects of source material. Chapter 3 explains the process of finding and stylizing drawing lines. Chapter 4 covers the solutions for the rest of the stylistic elements. Chapter 5 compares the output of this algorithm to the source material. Finally, chapter 6 elaborates on possible future improvements and chapter 7 gives an overview of the work done and concludes this thesis.

The terms used in this thesis are defined in the Glossary (Appendix I). The implementation of the algorithm and test scenes are in Repository and Additional Files (Appendix II), along with the source code. The system and software requirements to use the implementation and the user manual are in Requirements and User Manual (Appendix III). Images rendered by the devised algorithm are in Renders (Appendix IV).

2 Background

Graphic novels and comic books have been illustrated in a variety of styles *eg* Sin City[15], The Walking Dead[16], The Amazing Spider-Man[17] — all have very unique styles with their own stylistic elements. Illustrations in the *Water Memory* graphic novel present the following stylistic elements: grease pencil drawing lines, desaturated and tinted shading, pencil hatching, smudged color. Subchapter 2.1 explains these details in more depth.

The devised algorithm is implemented as an extension to the Maya's MNPR framework. Subchapter 2.2 gives more details about Autodesk Maya and explains features that are used by the MNPR framework. Using the features of Autodesk Maya, MNPR utilizes a 2-stage workflow for rendering. Subchapter 2.3 describes the 2-stage workflow of MNPR in more detail.

2.1 *La mémoire de l'eau (Water Memory)*

La mémoire de l'eau is a graphic novel written by Mathieu Reynès and illustrated by Valérie Vernay[18].

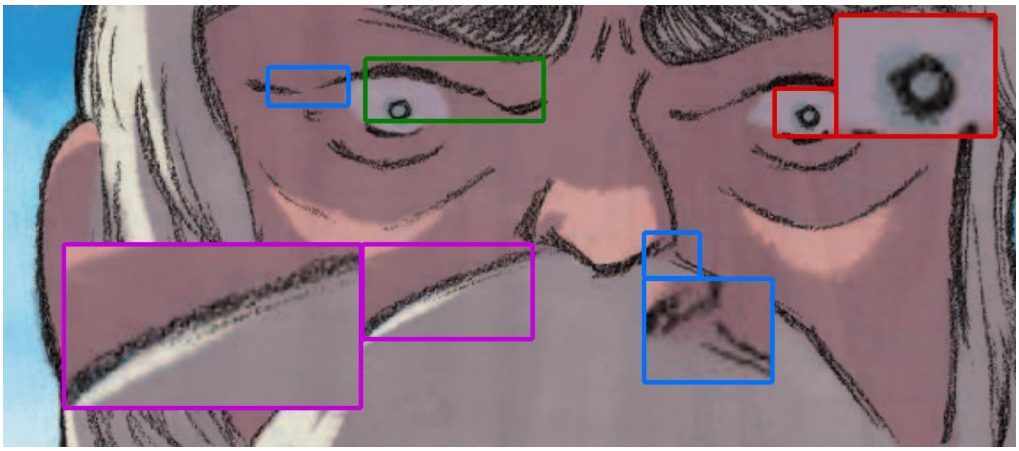


Figure 4. Panel from *Water Memory* graphic novel, which illustrates the stylistic elements of outlines and color smudging.

The illustration style of the graphic novel showcases outlines around details. One panel from the graphic novel is in Figure 4. The outlines in the panel look like they were drawn by hand and have varying thicknesses (green annotation). The intensity of the lines also

[15] https://en.wikipedia.org/wiki/Sin_City
[16] https://en.wikipedia.org/wiki/The_Walking_Dead_%28comic_book%29
[17] https://en.wikipedia.org/wiki/The_Amazing_Spider-Man_(comic_strip)
[18] *La memoire de l'eau* https://www.goodreads.com/book/show/35904894-water-memory

varies as the line annotated with green is more intense than the line annotated with purple. The outlines do not seem to be present around every detail and in some cases the lines cut out (blue annotations). Additionally, color smudging is visible around the character's eyes (red annotation). The pupils have circular outlines, but the color of the pupils is not confined to the circles. The light shading seems to be mostly flat and consists of two light levels: lit and in shade. In the shade the colors seem desaturated.

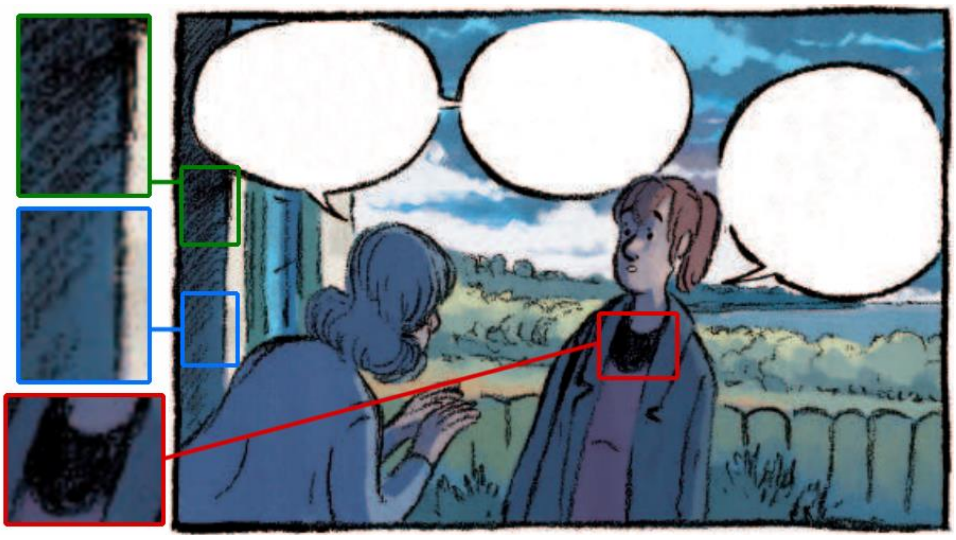


Figure 5. Examples of hatching and shade tinting in the *Water Memory* graphic novel.

In some cases, the surfaces in the shade are also tinted with a blue color. Figure 5 portrays an outdoors area, with dim daylight shining from the right side. The shaded wall of the building visible on the left is tinted blue (blue annotation) while the wall facing towards light is white. Additionally, hatching is used to convey the darkness of the shading or dark object colors. The wall in the shade (green annotation) is hatched with a pencil. The collar of the front-facing character (annotated with the red) has a heavy hatching texture.

Thus there are four specific stylistic elements that the rendering algorithm needs to achieve:

1. Controllable outlines with varying thickness and intensities.
2. Color smudging across outlines.
3. Flat light shading with color desaturation and tinting in the shade.
4. Hatching on dark-colored or shaded surfaces.

From the other panels of the comic it is also evident that these are the main contributors to the style in *Water Memory*. The devised approaches for all of these styles are described in Chapters 3 and 4.

2.2 Autodesk Maya

Autodesk Maya is a 3D computer graphics software developed by Autodesk. Maya provides tools for modeling, animation, simulation and rendering, etc. Maya is available for Windows, macOS and Linux operating systems[19]. Autodesk Maya has been used in the movie and video game industries for over two decades. Popular computer games such as *Mirror's Edge Catalyst*[20] (2016), *Deus Ex: Mankind Divided*[21] (2016), have utilized Maya for the production of assets[22]. Maya was also used to make animations in the game *Disco Elysium*[23,24]. From 1997 to 2015 Autodesk Maya had been used in the production of every film that had won the Academy Award for best visual effects[25].

Working in Autodesk Maya is based on virtual workspaces called *scenes*. Scenes contain objects, cameras, light information, etc. Scenes are rendered to viewports based on cameras. Using the viewport of Autodesk Maya, users see the rendering output or an approximation of it based on the 3D objects present in the scene (see Figure 6).

One of the features important to MNPR are materials that use shading methods described in an editor called ShaderFX[26] (ShaderFX materials). ShaderFX allows users to visually program how objects are rendered. The ShaderFX material renderings are output to either the Maya viewport and/or saved to buffers. ShaderFX allows the rendering to be output up to 8 buffers in addition to a 9th buffer containing information about the resulting fragment depth (the depth buffer). These 8 buffers can contain different information about the scene, such as surface normal vectors (normals buffer), light reflections in the scene (diffuse and specular buffers – lighting buffers), colors of objects (color buffer), etc. Buffers containing geometric data (normals and depth buffers) are called Geometry-buffers (G-buffers) [4]. The color buffer, lighting buffers and G-buffers in turn serve as the inputs to MNPR, which processes them to render the scene in a desired non-photorealistic style.

---

[19] https://www.autodesk.eu/products/maya/overview
[20] https://en.wikipedia.org/wiki/Mirror%27s_Edge_Catalyst
[21] https://en.wikipedia.org/wiki/Deus_Ex:_Mankind_Divided
[22] https://www.autodesk.com/campaigns/autodesk-for-games
[23] https://en.wikipedia.org/wiki/Disco_Elysium
[24] https://youtu.be/3f-AATwoM50
[25] *And the Oscar for best visual effects goes to…Autodesk's Maya* https://venturebeat.com/2015/01/15/hollywood-fx-pros-i-want-to-be-an-oscars-maya-winner/
[26] https://help.autodesk.com/view/MAYAUL/2016/ENU/?guid=GUID-EBC6DF48-857D-4230-9D3C-0B04DAF58403

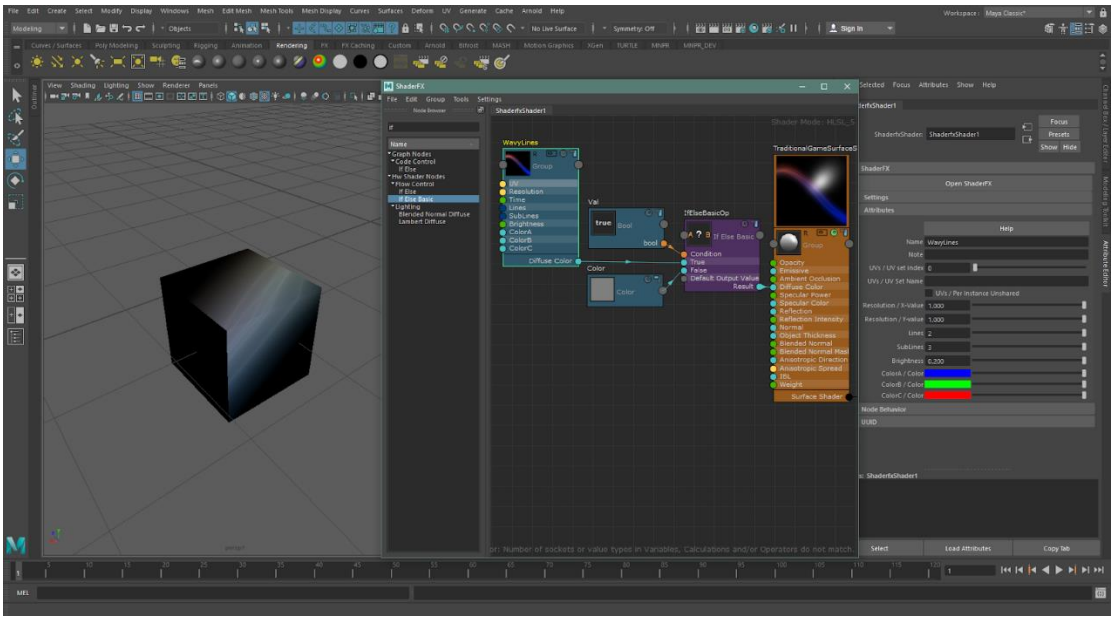


Figure 6. Autodesk Maya UI with a viewport on the left and the ShaderFX editor window.

Overall Maya provides a lot of different tools for different workflows. Users are also able to add functionality using plugins. One such plugin is MNPR.

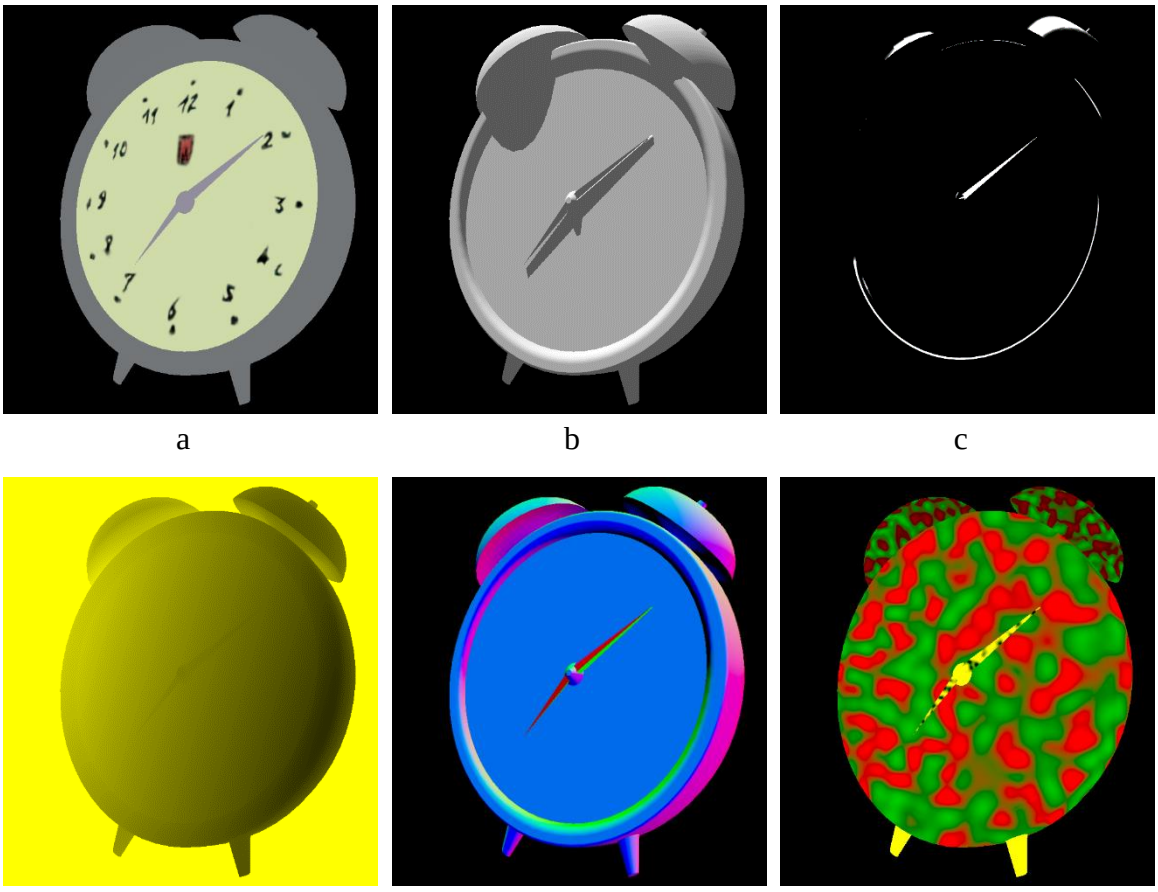
2.3 Maya Non-Photorealistic Rendering Framework (MNPR)

Maya Non-Photorealistic Rendering Framework is a plugin for Autodesk Maya[27], which provides a framework for non-photorealistic rendering. The framework works as a set of different stylization pipelines, which render the 3D geometry in Maya scenes in real-time. Each stylization pipeline consists of two stages: 1) rendering the Maya's ShaderFX materials in the scene to buffers (object-space rendering) and 2) rendering the output to the Maya viewport based on buffers received from the first stage (screen-space rendering).

First Maya renders the color, lighting, control buffers and G-buffers, then outputs these to MNPR. MNPR includes a set of ShaderFX materials, which configure Maya to output these 9 buffers (ShaderFX buffers): color, diffuse, specular, depth, normals buffers and a set of 4 control buffers. The 4 control buffers contain user-defined information about how to render different regions of the scene. Examples of such user-defined information are values that determine how dark or wide the outlines of objects are desired to be. Users define the control buffer information by painting the vertices of a 3D object mesh in the Maya viewport. The 4 control buffers provide up to 12 channels of floating-point values in total.

[27] https://mnpr.artineering.io/

Values stored in the 9 buffers are an input to the second stage of the MNPR stylization pipeline.



a b c

d e f

Figure 7. Color (a), diffuse (b), specular (c), depth (d), normals (e) buffers and an example of a control buffer (f).

The second stage works as a sequence of different algorithms. In addition to the 9 buffers from the first stage, MNPR also has access to additional textures and parameters in this stage. Example of an additional texture is a *substrate heightmap texture*, which defines the height and normal vectors of an underlying virtual canvas paper. MNPR utilizes the 9 buffers of data and its own data (from the additional textures and parameters) to derive more details. The new data is generated by algorithms that are run in parallel on every pixel of the screen. The algorithms store new information in new or already existing buffers. Among these buffers is an output buffer that is sent to the Maya viewport for displaying. For example, it is here that the color and depth buffers are used to detect edges.

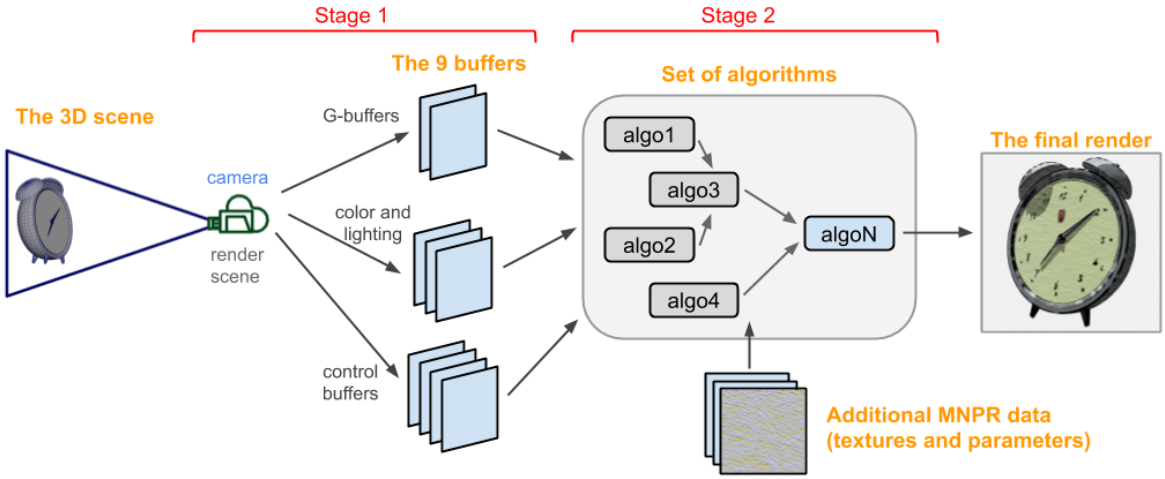


Figure 8. Overview of the 2-stage process of MNPR in Maya.

Knowing how MNPR works, it becomes clear that the devised algorithm (set of smaller algorithms) takes the 9 buffers of data from the 3D scene as input. Up to 4 control buffers are used for art-direction. Additionally, the algorithm has access to additional MNPR data textures. It is now possible to start creating the algorithm. The algorithm starts by processing the input data to render drawing lines.

3 Drawing Lines

Often NPR solutions emphasize visuals based on human perception [2]. Humans perceive and recognize object silhouettes [5]. Additionally, artists use drawing lines to convey the shape of objects [6]. Because of that, one prominent NPR visual is the outline around important objects.

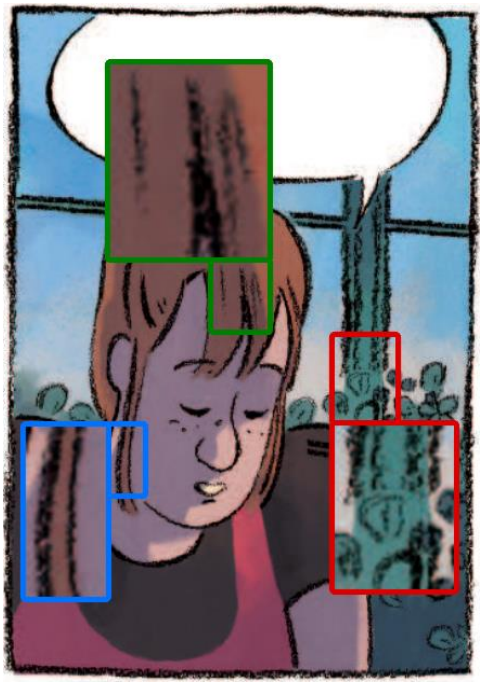


Figure 9. Drawing Lines example

In *Water Memory* the lines are drawn with a grease pencil (See Figure 9) and have varying thicknesses (blue annotation). These lines are not consistently present (red annotation) and appear to be applied with various different levels of pressure (green annotation). These drawing lines convey differences in depth, surface normal vectors and sometimes in color. Different solutions to rendering drawing lines have been previously proposed.

Solution by Grabli et al. offers a variety of different styles, but Solution does not work in real-time [7]. Solutions by Lee et al. offer lines with varying thicknesses, but rely on lighting or color information, not depth or surface normals [8,9]. Real-time watercolor stylization by Montesdeoca et al. utilizes vectors towards edges for color overlaps shading [10]. The approach described in this thesis utilizes these vectors to produce the drawing lines.

This drawing line algorithm works in two distinctive stages: first initial line locations need to be found, then stylized drawing lines are produced based on the initial line locations. Subchapter 3.1 describes the process of finding the initial line locations in the scene and subchapter 3.2 focuses on stylizing the initially found lines following the style of *Water Memory*.

3.1 Finding Initial Line Locations

Methods to find the areas in images at which pixel values differ are known as edge detection methods. These methods are widely known in the fields of digital image

processing and computer vision[28]. Edge detection methods focus on finding the exact location of lines. These exact locations appear as a thin few pixels wide areas [11]. From this point on, the edge-detected thin lines will simply be referred to as edges. In *Water Memory* drawing lines appear due to the differences in depth, surface normal vectors and sometimes color. It is possible to apply edge detection methods on that data since MNPR receives the color, depth and normals as buffers from ShaderFX and makes it possible to access these buffers. In this case, edge detection methods are suitable for locating lines in the scene, which are to be used for further stylization.

The edges are found using the difference of Gaussians method [12]. This method finds the second derivative of the values in the input buffer. The second derivative provides cleaner edges, unlike the Sobel operator, which finds the first derivative and is therefore noisier [13]. The noise difference is seen by comparing Figure 10 to Figure 11. The Sobel operator has found excessive edges in the color (red), depth (green) and normals (blue) buffers. The edges found using the difference of Gaussians method are sharper and thinner.

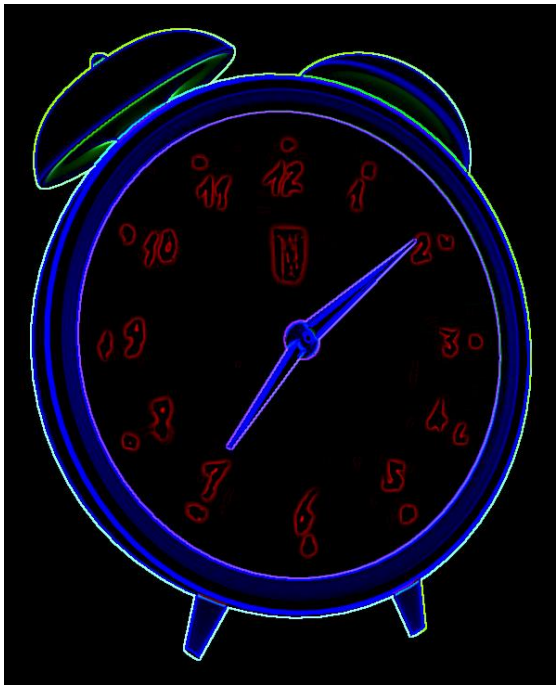


Figure 10. Edges found by the difference of Gaussians method.

Figure 11. Edges found by the Sobel operator contains additional noise.

The difference of Gaussians method is implemented by subtracting blurred image from a less blurred image. The images are blurred using Gaussian blur. Gaussian blur is

[28] https://en.wikipedia.org/wiki/Edge_detection

implemented by convolving the input buffer with a centered kernel, which values are calculated using the Gaussian function:

$$G_\sigma(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}.$$

The arguments x and y are the offsets in relation to center of the kernel. The value σ is the standard deviation, which sets the smoothness of the blurred image. In practice, the subtracted image is blurred with a k times the standard deviation of the image it is subtracted from. The value 1.6 is considered an optimal value for k to find edges [12]. The difference of Gaussians method is illustrated by the following formula:

$$DoG(x,y) = G_\sigma(x,y) - G_{k\sigma}(x,y).$$

The difference of Gaussians method is applied to the color, depth and normals buffers separately. However, the color and normals buffers contain data in 3-channels. The edge detection is applied to all channels separately and the maximum value across the 3-channels is considered as the output.

In case differences in values of depth, normals and color do not highlight outlines for specific objects, users need to have the option to manually declare that these objects have outlines. ShaderFX materials are configured to allow the user to set an object identification number (ObjectID) for each material. These ShaderFX materials are rendered to the 9 ShaderFX buffers. A channel of one buffer is set to display the ShaderFX materials as the ObjectID values. In practice the alpha channel of the normals buffer is used for the ObjectID values. Lines are detected by rendering the differences of the ObjectID values of neighboring pixels (see Figure 12). Differences are rendered as a value of 1, otherwise value 0 is used. The detected ObjectID-based lines are rendered into the vacant channel of the edges buffer. In practice the alpha channel is used.

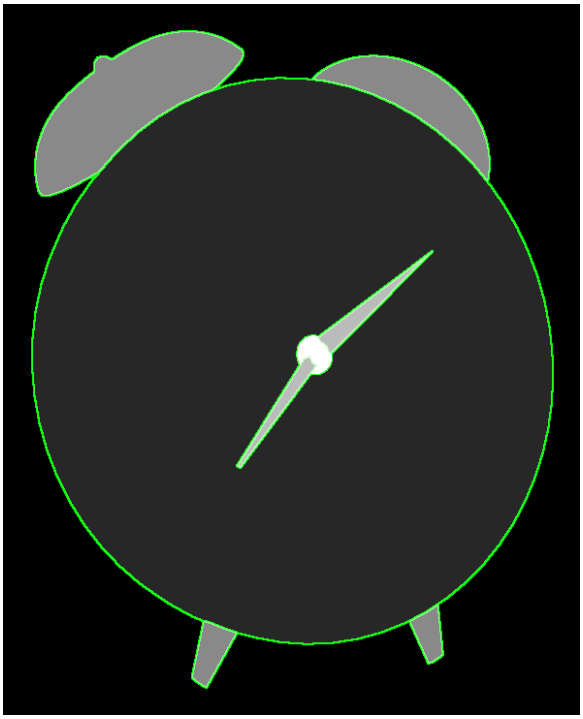


Figure 12. Edges (green) detected from different ObjectID values (grayscale).

At this point the edges buffer is considered to be a 2-dimensional array of quadruples. The 4 values represent the color channels and describe the edge intensities found based on color, depth, normals buffers and ObjectID values respectively. The array is illustrated by the following formula:

$$Edges = (color, depth, normals, oid).$$

The stylization process requires the line locations to be communicated as binary values, which label whether there is a line at a specific pixel. To achieve this, the values in $Edges$, which are provided by the edge-detection, are thresholded by a user-defined value $p_{threshold}$. Additionally, the control buffers are utilized to set how much the color-based edge-detection is taken into account prior to thresholding. The color-based edges are multiplied with the control values $c_{colorEdge}$. These control values are 0 by default, which means that the user needs to manually enable the color-based edges. This decision comes from the fact that in *Water Memory* lines conveying differences in color values are present less often than not. Afterwards, the maximum of the channel values for each pixel is used for thresholding. The result of thresholding is 0 if the maximum value is less than the threshold $g_{threshold}$, otherwise the result is 1. Finally, a single-channel $ThresholdedEdges$ buffer is created by rendering the thresholding results (see Figure 13). This process is described by the following formula:

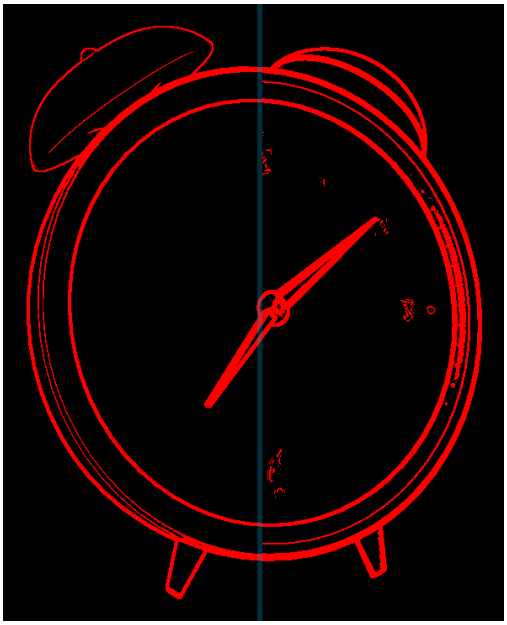


Figure 13. $ThresholdedEdges$ buffer: high threshold (left) and low threshold with art-directed use of colors (right)

$$Edges_{fromColor} = Edges_{fromColor} \cdot c_{colorEdge}$$

$$ThresholdedEdges = \begin{cases} 0, & \max(Edges_{color}, \dots, Edges_{oid}) < g_{threshold} \\ 1, & else \end{cases}.$$

The $ThresholdedEdges$ buffer is then used by the line stylization stage.

3.2 Stylizing as Drawing Lines

To allow users to art-direct the stylization of drawing lines, 2 types of control are provided: the thickness and the intensity of the drawing lines. Both of these controls are available through the use of control buffers and as global parameters. Since the drawing lines have varying thicknesses and intensities, the algorithm needs to know for every pixel the intensity of the drawing line, which the pixel is a part of, if at all. This data is found in five steps:

1. Thresholded edges are blurred.
2. Gradient vectors towards the closest edges are found.
3. The closest edge locations in the buffer are found using the gradient vectors.
4. The buffer containing closest edge locations is dilated.
5. Control buffers are dilated.
6. Local parameters are retrieved from the control buffers.

Similar approach has been described by Montesdeoca et al. for gaps and overlaps effect in the watercolor rendering algorithm in MNPR [10]. Unlike gaps and overlaps, this approach is used for the stylization of drawing lines and the local parameters are retrieved from the location of drawing lines.

As the first step, the thresholded edges buffer is blurred. This is done using Gaussian blur with the standard deviation of 10, which allows the edges to be blurred over a large enough area. The blurring is rendered to a single-channel $BlurredEdges$ buffer (see Figure 14). The higher values in the blurred edges buffer mean that an edge is nearby in the thresholded edges buffer, while the lower values imply the opposite.

Figure 14. Blurred edges buffer.

The $BlurredEdges$ buffer makes it possible to approximate the direction towards the closest edge location by finding the

gradient of the buffer. The directions are rendered to a dual-channel ∇B$lurredEdges$ buffer (see Figure 14). The implementation of this process is illustrated by the following formula:

$$\begin{aligned} dx(x,y) = {} & BlurredEdges(x+1,y) \\ & + 0.5 \cdot \big(BlurredEdges(x+1,y+1) + BlurredEdges(x+1,y-1)\big) \\ & - BlurredEdges(x-1,y) \\ & + 0.5 \cdot (BlurredEdges(x-1,y+1) + BlurredEdges(x-1,y-1)) \end{aligned}$$

$$\begin{aligned} dy(x,y) = {} & BlurredEdges(x,y+1) \\ & + 0.5 \cdot \big(BlurredEdges(x+1,y+1) + BlurredEdges(x-1,y+1)\big) \\ & - BlurredEdges(x,y-1) \\ & + 0.5 \cdot (BlurredEdges(x+1,y-1) + BlurredEdges(x-1,y-1)) \end{aligned}$$

$$\nabla BlurredEdges(x,y) = normalize(dx(x,y), dy(x,y)).$$

As the third step, the closest edge locations are found using the gradient vectors in the $\nabla BlurredEdges$ buffer. This is done by iteratively moving in the direction of the gradient vector. Before each iteration, the iterated position pos of the $ThresholdedEdges$ buffer is checked. If $ThresholdedEdges(pos) = 1$ then the iterated position is at an edge and the iteration process is ended. Otherwise the iterating continues while until an iteration steps limit of n is reached. In the implementation $n = 20$. After the iteration process, the results are rendered to $EdgeLocations$ buffer. If the iterating process ended with the iterated position being at an edge, the iterated position is rendered. Otherwise, an off-screen position is rendered, in practice $(-1,-1)$ is used. In the $EdgeLocations$ buffer, some pixels, which are close to an edge, do not hold the coordinates to the closest edge (see Figure 16). This is

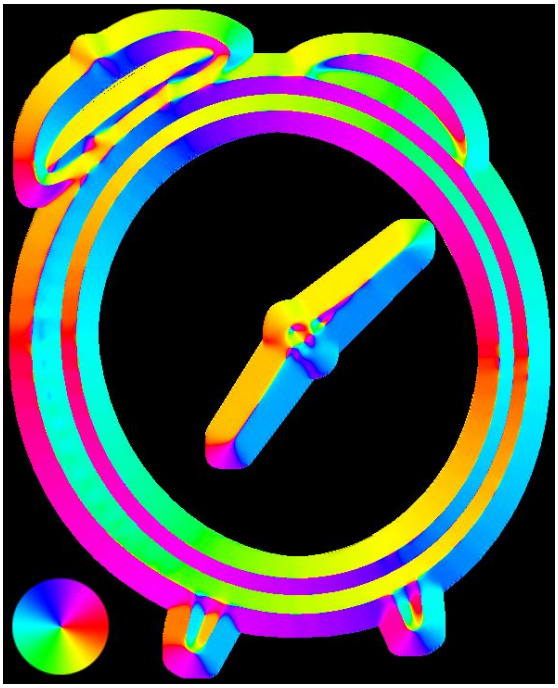


Figure 15. The gradient vectors in the $\nabla BlurredEdges$ buffer. The hue of the colors translates to vector directions, as displayed by the legend on the bottom left.

due to the $thresholdedEdges$ buffer holding disconnected edge pixels instead of continuous edges (green annotation in Figure 16 and Figure 17).

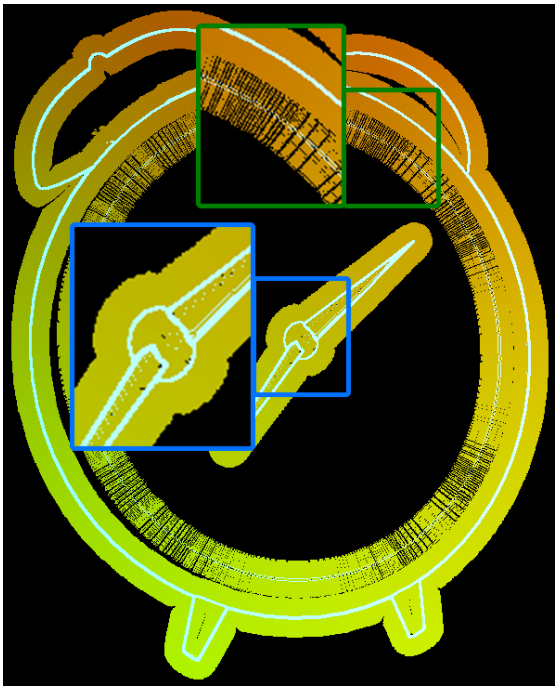


Figure 16. Closest edge locations are missing for some pixels, which are near an edge. Edges from $ThresholdedEdges$ are highlighted with a white color.

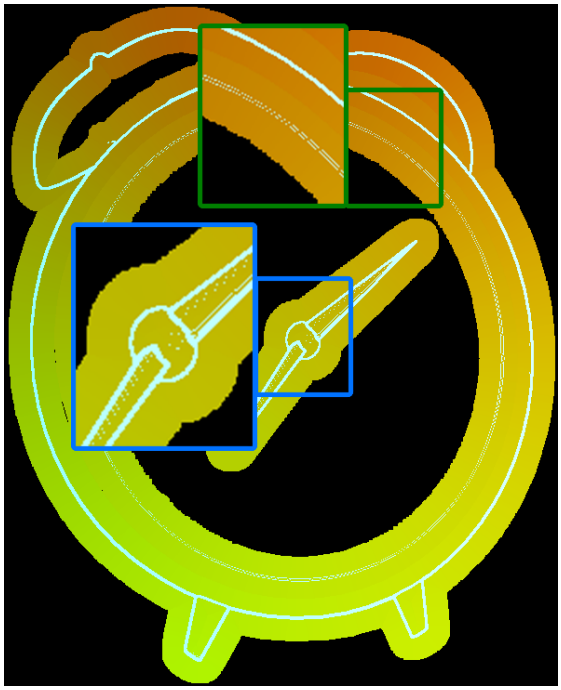


Figure 17. Closest edge locations are now present in the pixels close to an edge.

The $EdgeLocations$ buffer is dilated to solve the issue with some pixels not having the closest edge locations. The dilation is done with a centered 3×3 kernel and it outputs the edge location value closest to the center pixel. The dilation is done twice to get the result of dilating with a 5×5 kernel as this way less pixels need to be compared. In the implementation, the value $(-1,-1)$ is used as fallback in case no other values are found with the kernel. The result of the dilation process is visible in Figure 17. The method for choosing the value during dilation with the kernel K is illustrated by the following formula:

$$Distances = \{(x', y', distance) \mid (x', y') \in K \wedge (x', y') \neq (-1,-1)\}$$

$$output = \begin{cases} \min\limits_{distance}(Distances), & Distances \neq \emptyset \\ (-1,-1), & otherwise \end{cases}.$$

The control buffer values are not correct at the locations of the edges in the $ThresholdedEdges$ buffer. That is because half of the edges are around objects (see Figure 18). To solve this problem, the line control values for thicknesses $c_{thickness}$ and intensities $c_{intensity}$ are dilated. The dilation is done twice with a centered 3×3 kernel. The dilation

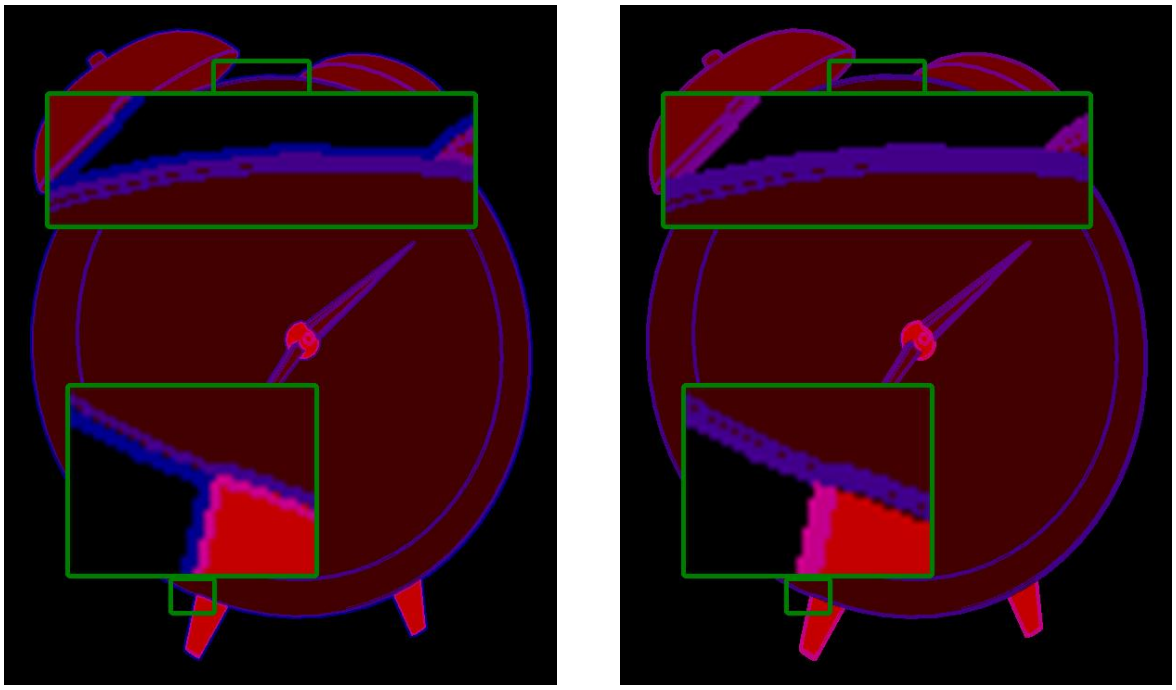


Figure 18. Control buffer before (left) and after (right) dilating. The red is line intensity control value, the blue highlights the location of edges from $ThresholdedEdges$ buffer.

process outputs the line control values, which originate from pixels with the least distance to the viewer. The line control values are taken from the pixels that are closer, because half of the edge around the object covers a surface, that is behind the object. The dilation process is illustrated by the following formula:

$$Controls = \{(c_{thickness}, c_{intensity}, depth) \mid (x', y') \in K \wedge depth = Depth(x', y') \wedge (c_{thickness}, c_{intensity}) = Control_{lines}(x', y')\}$$

$$output = \min_{depth}(Controls).$$

The result of dilation is shown in Figure 18. The line control values of an object are now set by the control values of the object and not by the surfaces behind the object. It is now possible to read the correct line control values and render the drawing lines.

Drawing lines are not rendered until the very last step in the overall algorithm. This is because, the drawing lines are applied on top of surface shading outputs (Surface Shading) The lines are rendered into the 4-channel stylization buffer, which contains the results from other algorithms described in Surface Shading. The rendering starts by reading the control values for the line thickness. The control values are retrieved from the closest edge location $e_{loc} \in EdgeLocations$. The thickness control value $c_{thickness}$ is then used to calculate the actual radius $line_r$ of the line by multiplying the control value with the global line thickness

parameter $p_{thickness}$. The distance between the pixel and e_{loc} is passed through a falloff function $falloff_{line_r,\ 0.1 \cdot line_r}(x)$. The falloff function is the following:

$$falloff_{range,start}(x) = \begin{cases} 1, & x < start \\ \max\left(0, \frac{range - x}{range - start}\right), & otherwise \end{cases}.$$

The falloff function is illustrated by Figure 19. The falloff function filters out pixels, which are outside of their corresponding line radius. If the returned value $line_{falloff} = falloff_{line_r,\ 0.1 \cdot line_r}(distance)$ is 0, then the pixel does not belong to any line. Otherwise, $line_{falloff}$ is used to determine the transparency of the pixel belonging to a line.

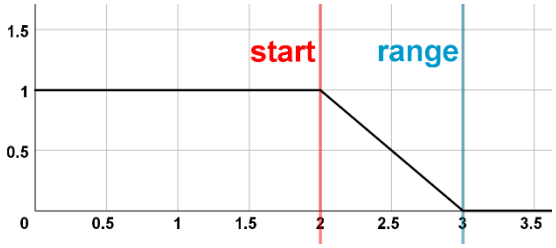


Figure 19. The falloff function $falloff_{3,2}(x)$.

The process of determining the transparency of a line pixel utilizes the control values for the line application pressure (line intensity). The line pixel intensity value is considered as the sum of the line intensity value $c_{intensity}$ and 1 is multiplied with the global line intensity parameter $g_{intensity}$. This value is used in calculating a threshold, which sets the lower boundary for the height of the paper (substrate height). The threshold is calculated as follows:

$$pixel_{intensity} = (1 + c_{intensity}) \cdot g_{intensity}$$

$$threshold = max\left(0, 1 - \frac{pixel_{intensity}}{2}\right).$$

The threshold is then subtracted from the multiplication of substrate height and the falloff value. If the subtraction value $substrate_{excess}$ is less than 0, then the line pixel is transparent. Otherwise the actual transparency of the drawing line is calculated by multiplying the subtraction value with $line_{intensity}$. The output pixel value is then rendered by linearly interpolating between the stylization buffer pixel color and black color by the value of transparency:

$$substrate_{excess} = substrate_{height} \cdot line_{falloff} - threshold$$

$$transparency = pixel_{intensity} \cdot substrate_{excess}$$

$$out_{color} = Stylization_{color} \cdot (1 - transparency) + black \cdot transparency.$$

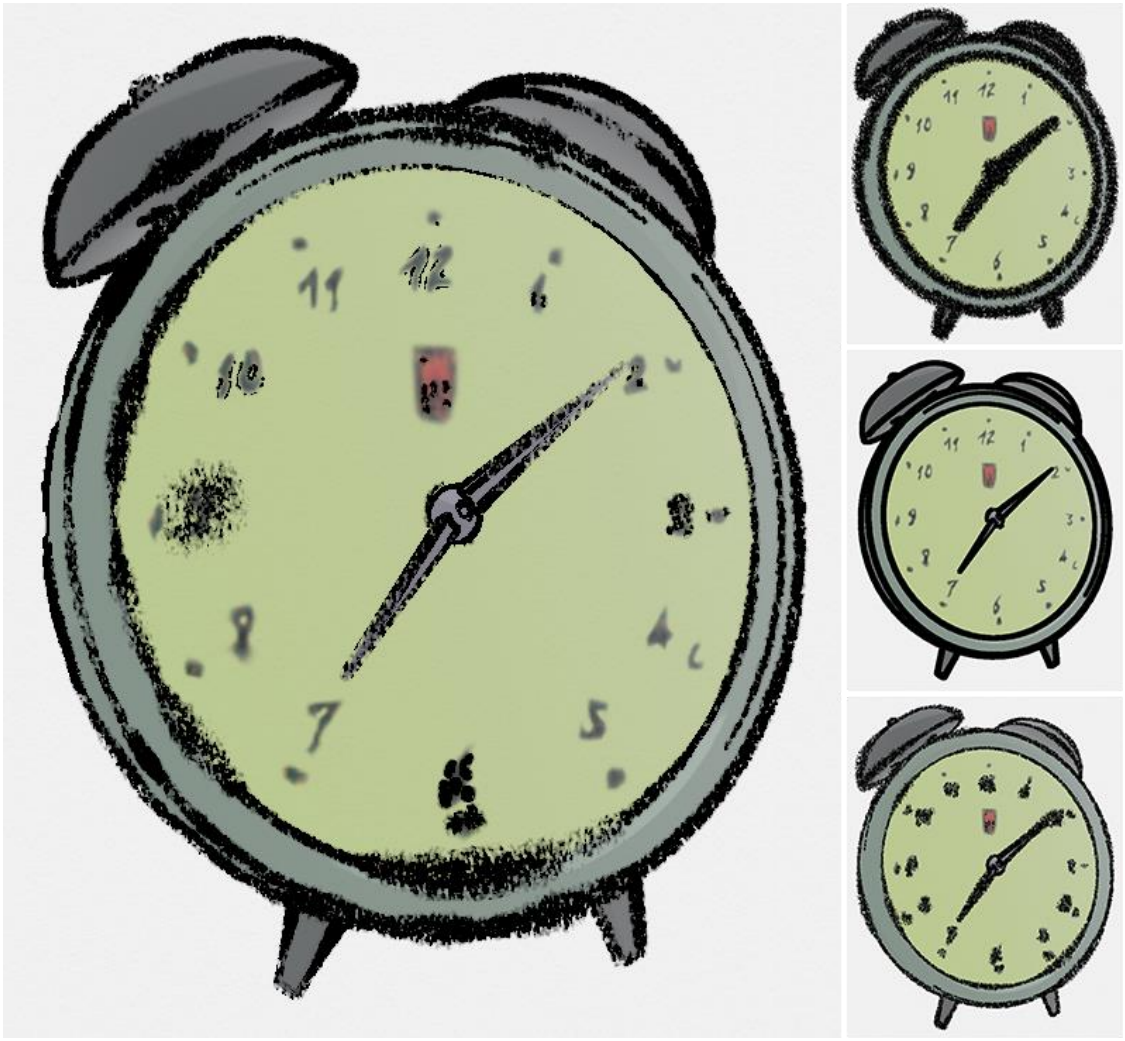


Figure 21. Example results of rendering drawing lines to the stylization buffer.

The resulting drawing lines have varying thicknesses and intensities (see Figure 21). Additionally, the substrate texture sets the texture of the drawing lines. Meanwhile the edge detection and thresholding along with the line thicknesses and intensities are controlled using art-direction controls (see Figure 20) and global parameters (see Figure 22). In addition to drawing lines, the surfaces of objects in the graphic novel are stylized as well.

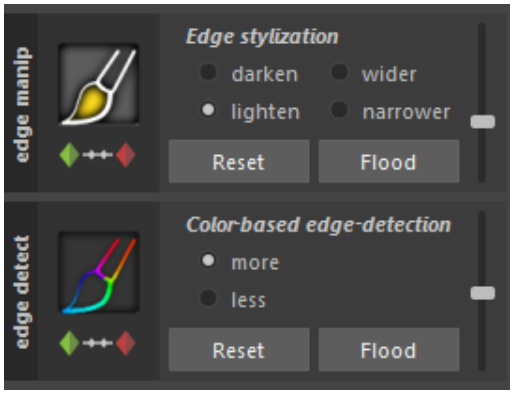


Figure 20. Drawing line controls.

Figure 22. Drawing line global parameters.

4 Surface Shading

This chapter considers surface shading as shading of object surfaces, unlike rendering of object outlines. It is common for non-photorealistic rendering methods to stylize object surfaces. One of the most common and basic surface shading method is included in *cel shading* [14]. Other surface shading methods include hatching of shaded areas, tinted shading or distortions such as color bleeding across surfaces.

The cel shaded surfaces have discrete light levels, which mimic the look of 2D-cartoons. However, the shading of surfaces is more complex in *Water Memory* (see Figure 23). The area annotated with blue showcases light-based shading, which has smooth transitions between discrete light levels. Surfaces with darker colors and areas in extreme shade have a hatching texture (red annotation). Additionally, color smudging is present near outlines (green annotation).

Figure 23. Examples of surface shading elements in *Water Memory* graphic novel.

Subchapter 4.1 describes the implementation of shading with discrete light levels along with shade tinting and desaturation of surfaces that appear to be in the shade. Subchapter 4.2 explains the color smudging effect and subchapter 4.3 focuses on hatching.

4.1 Discrete Shading

Figure 24. An example of shade tinting from the *Water Memory* graphic novel.

Figure 25. An example of surfaces being desaturated in shade, from the Water Memory graphic novel.

The light levels in the Water Memory graphic novel appear to be mostly discrete. The light levels do have smooth transitions, but the transitions are short and only occur at the edges of the light level. Additionally, the surfaces in the shade appear to be desaturated and tinted (see Figure 24 and Figure 25).

Figure 27. Diffuse buffer displaying an object lit by a red and white light.

In order to make the light levels in the scene discrete, the diffuse lighting buffer is required. Each pixel value in this buffer represents color of diffuse light, which is reflected from the surfaces in the scene (see Figure 27). These values need to be made discrete before they are applied to the scene. To make the light levels discrete, light amount (light intensity) is considered to be the *value*-channel in the HSV color model [15]. The value-channel of the HSV color model provides a value, which represents how much the color is mixed with black and ranges from 0 (black) to 1

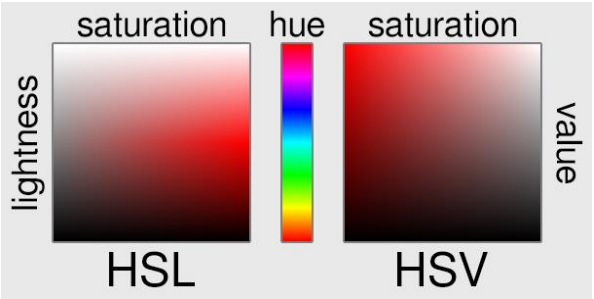


Figure 26. The HSL and HSV color models

(not mixed with black). Another hue-based color model – HSL has a channel for *lightness* instead of value and this represents how much the color is mixed with either black or white. The HSV color model is chosen over the HSL model as it is needed that the colorful lights maintain high light intensity. The use of HSL model would change the color of red light to white, which is not desired (see Figure 26).

Using the light intensity, the light in diffuse buffer is rendered as discrete. This is done by passing the light intensity through the following function:

$$\text{intensity}'(x) = \begin{cases} l_{high}, & x \in (t_4, \infty) \\ lerp(l_{high}, l_{mid}, \frac{x-t_3}{t_4-t_3}), & x \in (t_3, t_4] \\ l_{mid}, & x \in (t_2, t_3] \\ lerp(l_{mid}, l_{low}, \frac{x-t_1}{t_2-t_1}), & x \in (t_1, t_2] \\ l_{low}, & otherwise \end{cases}.$$

This function classifies the intensity into 3 levels while linearly interpolating (lerp) between the levels (see Figure 28). Values l_{high}, l_{mid} and l_{low} are user-defined global parameters, which set the discrete light intensity levels. Values t_1, t_2, t_3 and t_4 are calculated from 4 user-defined global parameters and these values set the thresholds, which decide the discrete light intensity level. The HSV value-channels of the initial light colors are set to equal the discrete light intensity and rendered to a $DiscreteDiffuse$ buffer (see Figure 29).

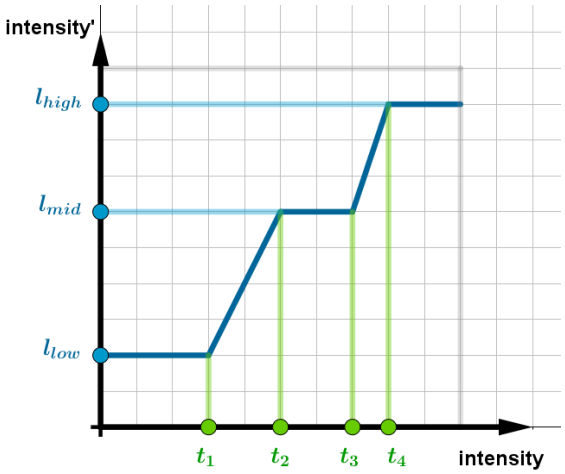


Figure 28. Illustration of the *intensity'* function.


Figure 29. The $DiscreteDiffuse$ buffer.

The shade tinting is done based on the discrete diffuse light values. The values from $DiscreteDiffuse$ buffer are rendered into the $TintedDiscreteDiffuse$ by mixing the discrete light colors with a user-defined color. The mix is a weighted sum of the discrete light

color and the tint color. The weight of the discrete light is 1, while the tint color weight w_{tint} is calculated as follows:

$$w_{tint} = (1 - DiscreteDiffuse_{intensity}) \cdot g_{tintWeight}.$$

The $DiscreteDiffuse_{intensity}$ is the intensity of the discrete light color in the HSV value-channel and $g_{tintWeight}$ is a user-defined parameter, which controls the amount of tint color applied to the shade. The resulting $TintedDiscreteDiffuse$ buffer is shown in Figure 30.

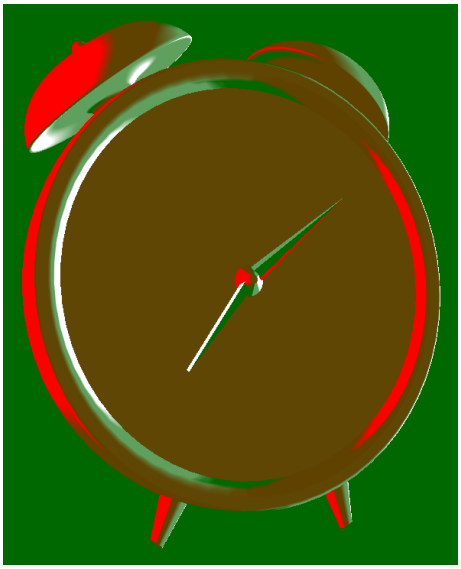


Figure 30. Colors in the $TintedDiscreteDiffuse$ buffer.

Next the surfaces are desaturated based on the $DiscreteDiffuse$ color intensities. The surface colors are taken from the $Stylization$ buffer, which MNPR sets to contains the colors of the surfaces. The desaturated surface color $surface'$ is found by changing the HSV color model saturation-channel value of the surface color. This process is described by the following formula:

$$desaturation = (1 - DiscreteDiffuse_{intensity}) \cdot (1 - g_{saturationWeight})$$

$$surface'_{saturation} = surface_{saturation} \cdot (1 - desaturation).$$

The value $g_{saturationWeight}$ is a user-defined parameter in the range [0,1]. This parameter allows the user to control how much desaturation is done on the surfaces in shade.

As the last step in discrete shading, the $TintedDiscreteDiffuse$ colors are applied to the desaturated surfaces. This is done by multiplying color values of $TintedDiscreteDiffuse$ and the color values of desaturated surfaces. The result is rendered into the $Stylization$ buffer (see Figure 31). Next, the shaded colors are used in the color smudging algorithm.

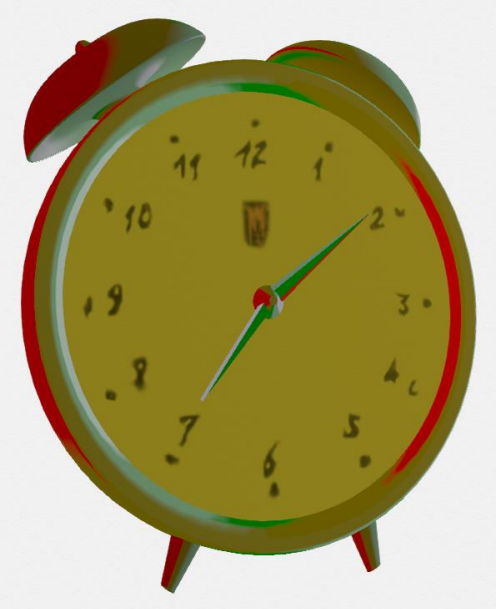


Figure 31. Desaturated and tinted surfaces in the $Stylization$ buffer.

4.2 Color Smudging

It is common for colors to spread in hand drawn images. This is especially noticeable if the color spreads across the outline borders of a drawn object. For example, this is the case with watercolors, which tend to mix with other colors across outline borders. Algorithms have been proposed to create that watercolor spreading effect [10,16].

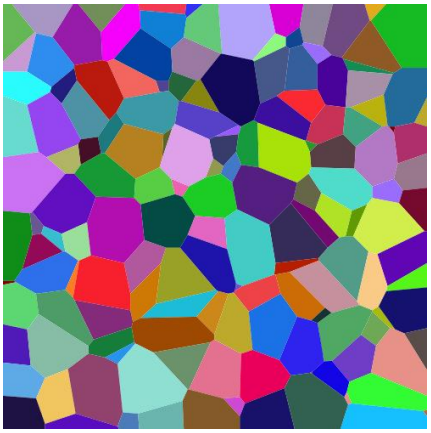


Figure 32. Voronoi diagram[29].

Figure 33. The output of the water color rendering by Chen et al. [16].

The approach by Chen et al. utilizes Voronoi diagrams (see Figure 32) to tile the image into cells and to bleed the colors of the tiled image. This approach causes colors to bleed all around the image and in both directions across the outlines (see Figure 33). In the case of *Water Memory*, the color smudging only happens around outlines and only in a single direction across the outlines.

The color overlapping method described by Montesdeoca et al. uses information found in the blurred edge buffer (similar buffer was described in Chapter 3) to locate nearby outlines. The nearby outlines are used to find the color of neighboring surfaces. These colors are then used to achieve the color bleeding effect by mixing the surface colors separated by outlines. In addition to color bleeding, this method also adds gaps in between colors, which is not desired in the case of *Water Memory* (see Figure 34). However, that algorithm can be modified to resemble the illustrations of *Water Memory*. Thus that algorithm by Montesdeoca et al. is chosen as a base for the algorithm devised in this subchapter.

[29] https://en.wikipedia.org/wiki/Voronoi_diagram

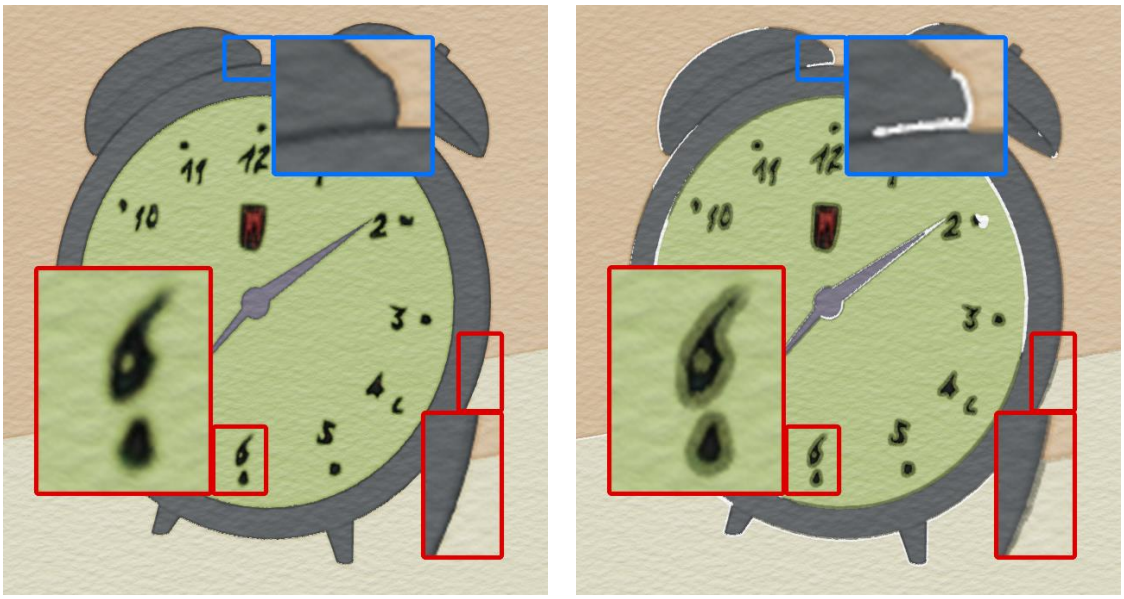


Figure 34. MNPR watercolor rendering without gaps and overlaps (left) and with gaps and overlaps (right).

Similarly to the gaps and overlaps algorithm, the color smudging is done by mixing surface colors with colors from neighboring surfaces. Since the line stylization algorithm (Chapter 3) is also based on the algorithm by Montesdeoca et al., the nearby outlines have already been found and are stored in $EdgeLocations$ buffer. The surface colors are taken from the $Stylization$ buffer, which contain the surface colors from the discrete shading algorithm. The location of a neighboring surface is taken by first finding a normalized vector towards the nearby outline. The normalized vector is then multiplied with a constant value and added to the location of nearby outline to get the location of the neighboring surface. The multiplication is done, because the outline might have a thickness of multiple pixels (see Figure 35). The neighboring surface color $color_{neighbor}$ is then picked from the $Stylization$ buffer and used in the mixing process.

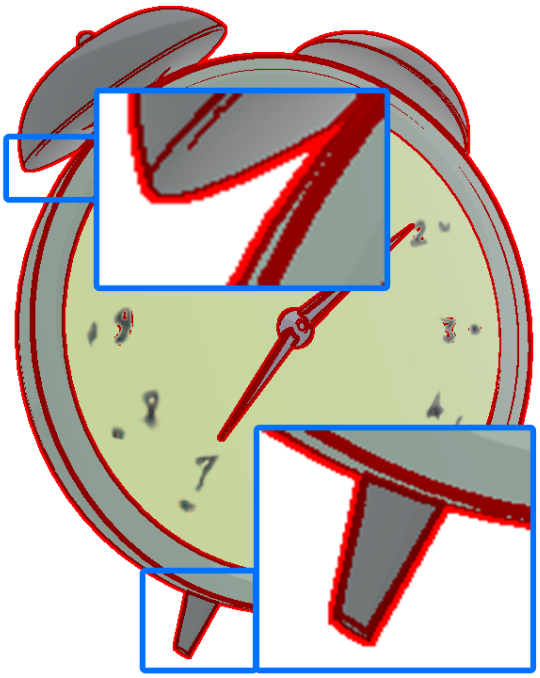


Figure 35. Outlines in $ThresholdedEdges$ buffer being applied to the $Stylization$ buffer. The outlines outside the object are bright red while the outlines on top of the object are dark red

The color smudging locations are art-directed by the user using control buffers. The control buffers contain parameters for color smudging range c_{range} and for the falloff start $c_{falloff}$ of color mixing intensity relative to the smudging range. Both art-directed control values are multiplied with their corresponding global control parameters. The global control values and art-direction controls are shown in Figure 36. While the color spreading distance is an absolute value, the value of the smudging range parameter is signed.

Figure 36. Smudging global parameters and art-direction controls.

The sign of c_{range} determines if the neighboring surface color needs to originate from a surface that is further away from the viewer (positive value) or closer to the viewer (negative value). The depth buffer is used to determine this. Additionally, users are provided with a global parameter, which sets the minimal required depth difference between the neighboring surfaces. This parameter is used to let the user avoid color smudging on surfaces with too small depth differences. In case the neighboring surface depth does not match up to the sign of c_{range} or the depth differences are lower than the minimal allowed difference, the surface color is not smudged. Otherwise the mixing intensity is found.

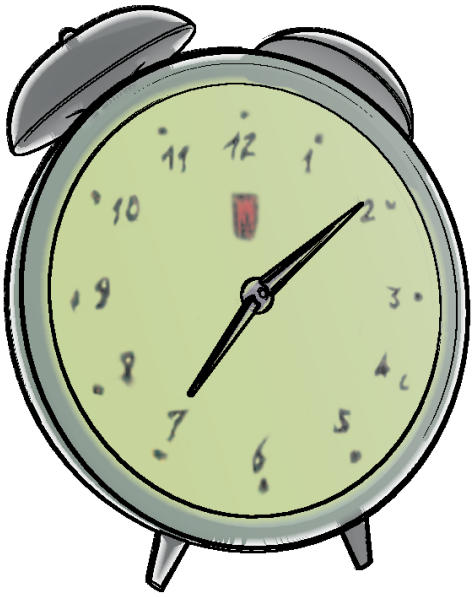


Figure 37. The output of color smudging.

The mixing intensity falloff c_{range} determines from how far the linear falloff of the mixing intensity starts. First, the screen-space distance d between the pixels of neighboring surfaces are found. The intensity of mixing and the resulting color $smudged$ from the mixing process is calculated as follows:

$$intensity = falloff_{c_{range},\ c_{falloff}}(d)^{falloffSpeed}$$

$$smudged = intensity \cdot color_{neighbor} + (1 - intensity) \cdot color_{local}.$$

The value of $falloffSpeed$ is defined by the user and it represents the speed of intensity falloff. The value of $color_{local}$ represents the local pixel in $Stylization$ buffer prior to being smudged. The smudged surfaces are output to the $Stylization$ buffer (see Figure 37).

4.3 Hatching

It appears that the surfaces that have a dark enough color or that are in dark enough shade are hatched with a pencil (see Figure 38). This shading does not seem to convey significant direction or spatiality. Therefore, it is valid to apply the hatching in image-space. Color, diffuse and specular buffers are required to find the locations where hatching should appear. The hatching texture is determined by the substrate texture.

Figure 38. Examples of hatching from *Water Memory* graphic novel.

It is worth noting that the lightness of hatching is not consistent and varies across surfaces with the same dark color. In Figure 38 it is visible that the hatching cuts out completely in the area annotated with red. This may be because of high specular light in that area.

To simulate the hatching in color-wise and light-wise dark areas, color lightness l_c and diffuse lightness l_d are considered to be the lightness of how strongly the pencil is used for hatching. The values of l_c and l_d are the HSV value-channels of the surface color in the color buffer and the diffuse color in the diffuse buffer. Both lightness values are divided with global parameters g_c and g_d respectively. Afterwards the minimum of 1, l_c and l_d is considered the actual lightness l, which is used for hatching. The following formula illustrates the process of deriving the lightness value l from l_c and l_d:

$$l = \min(1, \frac{l_c}{g_c}, \frac{l_d}{g_d})$$

The global parameters g_c and g_d serve as thresholds, which normalize values below the threshold to the range [0,1] and filter out too high lightness values by increasing them higher than 1.

The height h of the substrate is used to deviate the intensity of hatching in order to give the hatching a substrate texture. To allow the user to art-direct the intensity and location where the hatching happens, the control buffers contain a parameter c_{hatch}. This parameter value is increased by 1, because the value is 0 by default and to bring it to the range [0,2]. Having c_{hatch} in this range allows it to be used as the multiplier for substrate height h. The intensity of hatching is then found as follows:

$$strength = h \cdot c_{hatch} - l$$

$$intensity = (1 - specular) \cdot strength$$

The strength of how strongly the pencil is pressed against the paper is found by subtracting the lightness l from the multiplication of c_{hatch} and h. The intensity of hatching is then found by accounting for the specular lighting. The value of $specular$ is the maximum value across RGB channel values of the color taken from specular buffer. In practice the intensity is additionally multiplied with 2.5 to make the hatching more intense.

As the last step, the intensity of hatching is used to apply the hatching to surface colors in the $Stylization$ buffer. This is done by linearly interpolating between the values of the surface color and black color using the clamped value of hatching intensity. The result of hatching is shown in Figure 39.

Figure 39. The result of hatching. Some areas of the hatching are made lighter using art-direction controls.

5 Results

The criteria of assessment for non-photorealistic rendering algorithms, which mimic handmade graphics, is direct comparison with the source material [2]. Therefore, the success of this algorithm is evaluated by comparing images rendered with the algorithm to the illustrations in *Water Memory*. Two panels were selected from the graphic novel and recreated as scenes in Autodesk Maya.

The first selected panel is shown in Figure 40. This panel was selected as it showcases drawing lines with various thicknesses and intensities. Additionally, the panel contains color smudging and shade hatching. The drawing lines that appear in this panel feature cut-outs and mostly convey differences in depth and normals.

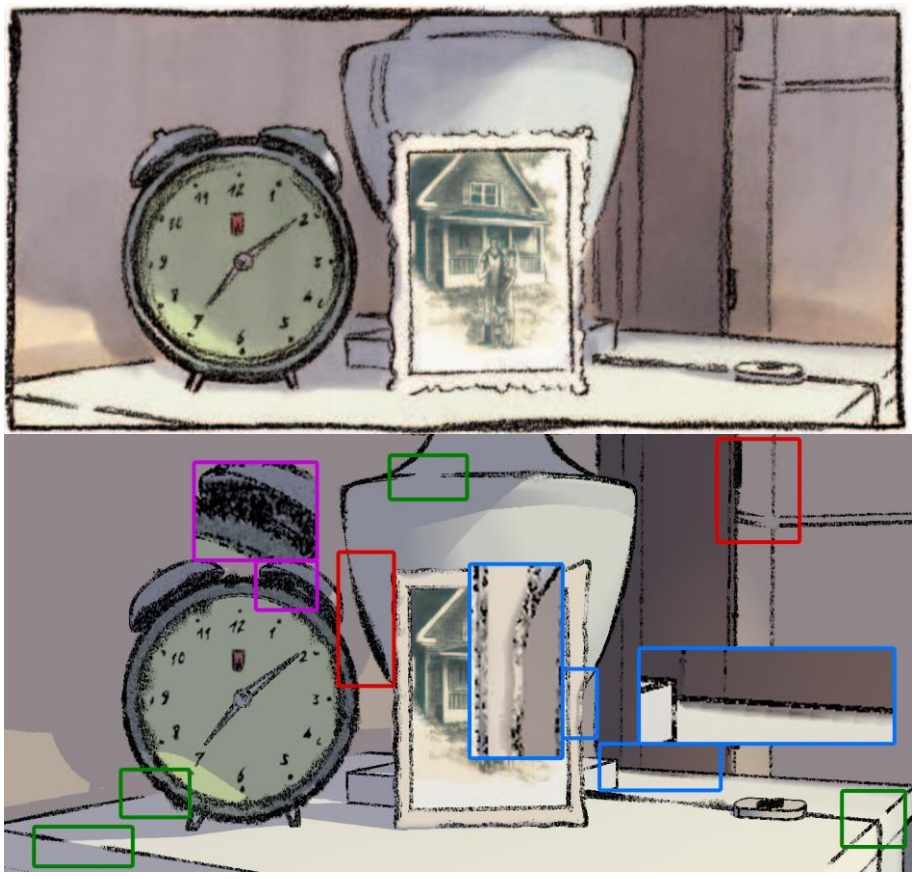


Figure 40. The first chosen panel from *Water Memory* (above) and the corresponding rendering result (below).

From the rendering output (see Figure 40) it is visible that the drawing lines have varying thicknesses and intensities (red annotation). The drawing lines cut out in the same areas as in the source panel (green annotation). The areas in the shade are tinted with a purple-blue

color. Color smudging is also displayed in the areas annotated with blue. Hatching is present as well (purple annotation).

The second panel is shown in Figure 41. This panel was chosen as it showcases the discrete surface shading and color-based hatching. Additionally, the areas in the shade are tinted and there are outlines that highlight differences in color.

Figure 41. The second chosen panel from *Water Memory* (above) and the corresponding rendering result (below).

In Figure 41 it is visible that the shading is discrete with smooth transitions between the discrete light levels. It is also visible that the shade is tinted with a light-purple color. The red annotation points out an example of drawing lines being rendered at the location of color differences. Meanwhile the green annotation shows the color-based hatching.

Results show that the four main stylistic elements of Water Memory are present in the rendering outputs of the devised algorithm. Both renders are in included in Appendix IV without the annotations.

6 Future Work

The devised algorithms for rendering the four main stylistic elements could be further improved. For example, dilation could be used instead of iteratively finding the closest edge location using the gradient vectors of blurred edges buffer. This approach removes the errors, which occur due to inaccurate gradient vectors towards the closest edges. This is done by setting an edge location for every pixel, at the location of which there is a thresholded edge. After this the edge locations buffer is repeatedly dilated by choosing the closest edge location among the values in the dilating kernel. The implementation of stylizing drawing lines using the dilation method is included as a separate style (*Water Memory Alt*) in the modified version of MNPR included with this thesis (Appendix II).

Furthermore, in case of outlines with varying thicknesses, this does not always help produce the correct output. The most potential edge location, which would include the pixel in the drawing line should be found instead of the closest edge location. The potential for being included in the drawing line of an edge location could be calculated by accounting for the line thickness and the distance between the pixel and the edge location.

In addition to difference of Gaussians and the Sobel operator, there are other means for detecting edges in scene. The edge-detection part of the drawing line rendering algorithm is the basis for the drawing line stylization. Despite that, the edge-detection method should be easily changeable. Some other method could be used for finding more suitable edges.

Although the hatching algorithm is not based on any specific algorithm for hatching, there are algorithms, which have been proposed before. One such hatching algorithm has been proposed by Praun et al. [17]. Other hatching algorithms could be used instead of the devised algorithm.

The current implementation of discrete light shading only allows the user to define 3 discrete light levels. The discrete light shading operates using 7 parameters in total. Users could instead declare the discrete light levels using the Maya Ramp UI element[30] (see). This would allow declaring more or less discrete light levels and more intuitive control.

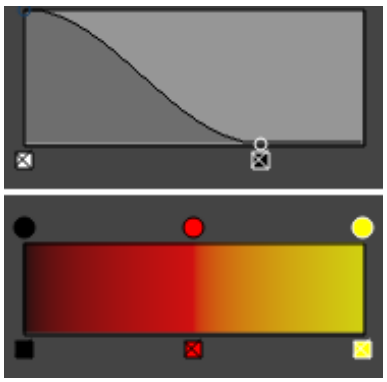


Figure 42. Maya Ramp attributes: curve (above) and color (below) ramps.

[30] https://help.autodesk.com/view/MAYAUL/2016/ENU/?guid=__cpp_ref_class_m_ramp_attribute_html

7 Conclusion

The goal of this thesis was to devise a real-time non-photorealistic rendering algorithm based on the graphic novel *La mémoire de l'eau (Water Memory)*. The graphic novel features stylistic elements such as outlines with varying thickness, color smudging, pencil hatching, flat light shading, color desaturation and tinting in the shade. These are also the stylistic elements that the algorithm is tasked with achieving.

The difference of Gaussians method was used for locating initial edges, which were then processed and used to render drawing lines with varying thicknesses and intensities. The gaps and overlaps algorithm designed by Montesdeoca et al. [10] was used in the processing of initial edge locations and for color smudging. The Hue-Saturation-Value color model was used for determining and setting the light intensities in the process of flat shading and for determining the color intensities for hatching.

The rendered elements of this algorithm were made to be art-directed and globally controlled by the user. Art-direction allows control over elements such as the line thickness and intensity, color contribution to edge-detection, the depth-wise direction and falloff of color smudging and the intensity of hatching. Global control parameters enable setting the light intensities of different levels of flat shading and global control over art-directed elements.

The success of this algorithm was assessed by comparing its output to the source material. For this purpose, two panels from the graphic novel were recreated as scenes in Autodesk Maya and rendered using the algorithm implementation in MNPR. While it was difficult to exactly recreate the hand drawn panels due to perspective and unclear sources of light, the main stylistic elements were clearly visible in the output of the rendering algorithm. Therefore, this algorithm succeeds in outputting renditions stylistically similar to the source material.

Special thanks go to the supervisor Raimond-Hendrik Tunnel, who helped with staying on track and gave suggestions to improve the thesis whenever asked, and Santiago Montesdeoca, who helped with implementing the algorithm in MNPR. Thanks go to Jaanus Jaggo, who provided the initial mesh for the scenes used in Results.

References

[1] Bruce Gooch and Amy Gooch, *Non-Photorealistic Rendering*, 1st ed.: AK Peters Ltd, 2001.

[2] Thomas Strothotte and Stefan Schlechtweg, *Non-Photorealistic Computer Graphics*.: Morgan Kaufmann, 2002.

[3] Santiago Montesdeoca et al., "MNPR: A Framework for Real-Time Expressive Non-Photorealistic Rendering of 3D Computer Graphics," 2018.

[4] Takafumi Saito and Takahashi Tokiichiro, "Comprehensible Rendering of 3-D Shapes," 1990.

[5] Jay Friedenberg and Gordon Silverman, *Cognitive Science: An Introduction to the Study of Mind*, 3rd ed.: Sage Publications, 2015.

[6] Forrester Cole et al., "Where Do People Draw Lines?," 2008.

[7] Stéphane Grabli, Emmanuel Turquin, Frédo Durand, and François Sillion, "Programmable rendering of line drawing from 3D scenes," 2010.

[8] Yujin Lee, Lee Markosian, Seungyong Lee, and John Hughes, "Line drawings via abstracted shading," 2007.

[9] Henry Kang, Seungyong Lee, and Charles Chui, "Coherent line drawing," 2007.

[10] Santiago Montesdeoca et al., "Edge- and substrate-based effects for watercolor stylization," 2017.

[11] Holger Winnemöller, Jan Eric Kyprianidis, and Sven Olsen, "XDoG: An eXtended difference-of-Gaussians compendium including advanced image stylization," 2012.

[12] David Marr and Ellen Hildreth, *Theory of Edge Detection*.: Royal Society, 1980.

[13] Rafael Gonzales and Richard Woods, *Digital Image Processing*, 3rd, Ed.: Pearson, 2008.

[14] Raul Luque, "The Cel Shading Technique," 2012.

[15] Max K Agoston, *Computer Graphics and Geometric Modelling*.: Springer, 2005.

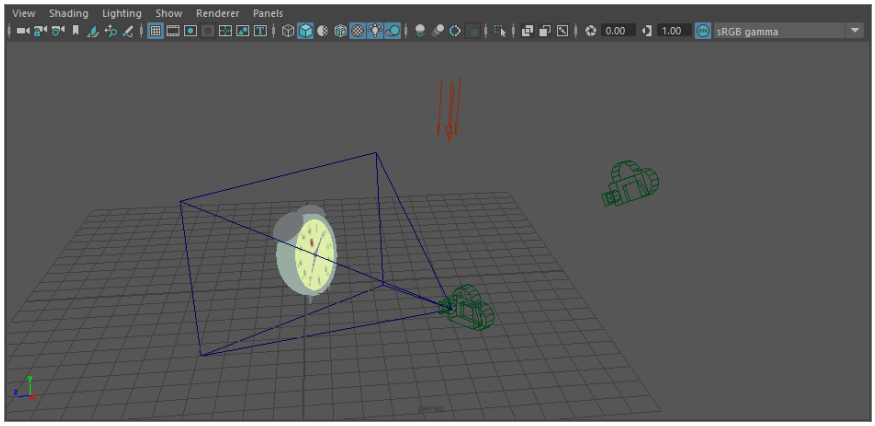
[16] Jiajian Chen, Greg Turk, and Blair MacIntyre, "Watercolor inspired non-photorealistic rendering for augmented reality," 2008.

[17] Emil Praun, Hughes Hoppe, Matthew Webb, and Adam Finkelstein, "Real-Time Hatching," 2001.

Appendix

I. Glossary

1. **Viewport** – The Maya UI element, which displays the open scene:



2. **Buffer** – 2D array containing 1-4 dimensional vectors of values. These values can be floating-point numbers or integers, signed or unsigned.
3. **Substrate** – the base material that is drawn on (eg paper). In MNPR, the substrate texture holds 2D normals (with the basis perpendicular in relation to the viewer) and the heightmap of the substrate.
4. **Heightmap** – a buffer or a part of buffer containing height values. Lower values correspond to lower altitudes while higher values correspond to higher altitudes.
5. **Depth buffer** – a buffer containing depth values for every pixel in the scene. Higher values correspond to steeper depths while lower values correspond to shallower depths.
6. **Fragment** – a rasterization position in the scene, which corresponds to a screen pixel. The fragment depth is the distance from the viewer.
7. **Diffuse and specular** – diffuse and specular are reflections of light from surfaces. Diffuse reflections are considered to be the reflections of light that scatter around in many directions. Specular reflections indicate the light reflections that reflect directly towards the viewer. Diffuse and specular light reflections are simply called diffuse and specular light in this thesis.
8. **Art-direction** – the process of manually configuring the stylization using control buffers. This is done in MNPR using *PaintFX*, which allows users to paint vertices of mesh with values that are rendered into control buffers.

9. **Painting of vertices** – giving vertices of 3D mesh certain values by painting on them in the viewport of Autodesk Maya.
10. **Screen-space** – the 2D coordinate system in which the horizontal axis corresponds to the width of the screen and vertical axis corresponds to the height of the screen. The height and width are in pixels.
11. **Clamping** – the process of changing a value as illustrated by the following formula follows:

$$value' = \min(1, \max(0, value)).$$

II. Repository and Additional Files

Repository link – https://github.com/Yleroimar/3D-Comic-Rendering

Scenes – This contains the scene files used in Results and all necessary assets, which are required for the scene to work properly. The contents of this folder need to be moved to Maya's project folder (*eg* C:\Users\[Username]\Documents\maya\projects\[project]).

MNPR_extended – This contains the MNPR files and the changes and implementations done for the devised algorithm. The files are to be moved to any desired location. Among the files is the file *insall.mel*, which needs to be dragged and dropped into the viewport of Autodesk Maya. Now MNPR installs and after installation Maya needs to be restarted. Installation does not differ from the installation of regular MNPR, which is found here: https://mnpr.artineering.io/installation

III. Requirements and User Manual

Requirements:

- Windows 10
- Autodesk Maya 2019.2

User Manual for opening test scenes:

Once MNPR_extended (Appendix II) has been added to Autodesk Maya, the substrate textures need to be downloaded:

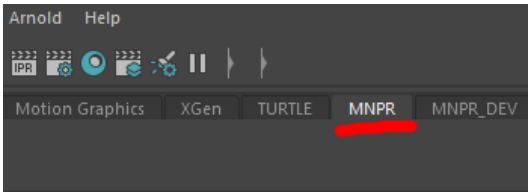


Figure 43. MNPR shelf tab.

1. Open Autodesk Maya.
2. Go to the MNPR shelf tab (Figure 3).
3. Right click on MNPR button on the shelf (Figure 44).
4. Click Download *MNPR substrates* and let it download.

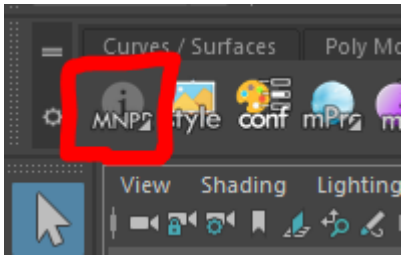
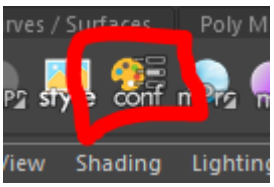


Figure 44. MNPR button on the shelf.

Once the downloading has been completed, the test scenes can be opened:

1. Press *File* on the menu bar.
2. Press *Open scene*.
3. Navigate to the projects folder where the contents of Scenes folder was placed (Appendix II).
4. Navigate into the *scenes* folder.
5. Double-click either *wm_scene1.ma* or *wm_scene2.ma*.
6. Once the scene opens, open *NPR configuration node* on the MNPR shelf:



7. Change any of the slider values and press *CTRL+Z* to undo the change. This has to be done as in some cases the saved configuration is not loaded automatically.

The scene should be open now and everything should be displayed as intended.

IV. Renders

Figure 45. Rendering results of *Water Memory* panels, which were recreated in Maya.

V. License

Non-exclusive licence to reproduce thesis and make thesis public

I, Oliver Vainumäe,

(author's name)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   3D Comic Rendering

   (title of thesis)

   supervised by Raimond-Hendrik Tunnel.

   (supervisor's name)

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Oliver Vainumäe

10/05/2020